# La machine abstraite KGRAM et son langage GRAAL pour l'interrogation de graphes de connaissances

**Olivier Corby — Catherine Faron-Zucker**

*INRIA, Edelweiss*
*2004 route des lucioles - BP 93*
*FR-06902 Sophia Antipolis cedex*
*olivier.corby@sophia.inria.fr*

*I3S, Université de Nice Sophia, CNRS*
*930 route des Colles - BP 145*
*FR-06903 Sophia Antipolis cedex*
*catherine.faron-zucker@unice.fr*

RÉSUMÉ. *Dans cet article nous présentons la machine abstraite KGRAM et son langage GRAAL pour l'interrogation de graphes de connaissances. KGRAM et GRAAL sont le résultat d'un travail d'abstraction que nous avons mené afin de proposer une solution générique au problème de l'interrogation de graphes de connaissances de différents modèles. Ce faisant, nous avons identifié des primitives abstraites de haut niveau permettant de répondre à ce problème ; elles constituent les expressions du langage GRAAL et les interfaces que manipule KGRAM pour ses structures de données et ses opérations de graphes. Nous décrivons la syntaxe abstraite de GRAAL et sa sémantique opérationnelle en Sémantique Naturelle et nous définissons KGRAM comme l'interprète de GRAAL, c'est-à-dire l'implémentation des règles de sémantique naturelle de GRAAL.*

ABSTRACT. *In this paper we present the KGRAM abstract machine and its language GRAAL dedicated to the query of knowledge graphs. KGRAM and GRAAL together result from an abstraction process we performed reach a generic solution to the problem of querying knowledge graphs in various models. We identified high level abstract primitives that constitute the expressions of the GRAAL language and the interfaces of the KGRAM abstract machine for both its data structures and its operations. We describe the abstract syntax of GRAAL and its operational semantics in Natural Semantics and we define KGRAM as an interpret of GRAAL that implements the inference rules of the semantics of GRAAL.*

MOTS-CLÉS : *Graphes de connaissances, Langage de requêtes, Machine abstraite, Sémantique naturelle*

KEYWORDS: *Knowledge graphs, Query language, Abstract machine, Natural Semantics*

Langages et Modèles à Objets – 2010, pages 1 à 15.

## 1. Introduction

Dans cet article nous présentons la machine abstraite KGRAM (acronyme pour Knowledge Graph Abstract Machine) et son langage GRAAL (acronyme pour GRAph Abstract query Language) pour l'interrogation de graphes de connaissances quelconques. KGRAM et GRAAL sont le résultat d'un travail d'abstraction que nous avons mené afin de proposer une solution générique au problème de l'interrogation de graphes orientés étiquetés et plus particulièrement de graphes de connaissances de différents modèles. Ce travail apporte un élément de réponse dans les enjeux actuels importants, liés à la multiplicité des langages de représentation des connaissances qui coexistent. Avec GRAAL et KGRAM, nous proposons d'unifier les mécanismes de raisonnement pour interroger des bases de connaissances de différents modèles. Pour ce faire, nous avons identifié des primitives abstraites de haut niveau qui constituent les expressions du langage GRAAL et les interfaces que manipule KGRAM. De fait, KGRAM est conçue comme un interprète de GRAAL et ne manipule que des interfaces aussi bien pour ses structures de données que pour ses opérations de graphes.

Le travail d'abstraction que nous avons mené pour définir GRAAL a été alimenté par les résultats du projet GRIWES [BAG 08] auquel nous avons participé. La conception de KGRAM a bénéficié de notre expérience passée de conception et de développement du moteur sémantique Corese[1] : Corese est dédié à l'interrogation de données RDF, RDFS et OWL Lite dans le langage SPARQL et ses structures de données internes et leur manipulation reposent sur le modèle des graphes conceptuels [COR 04, COR 07]. Enfin, par notre idée de construire une machine abstraite pour des graphes de connaissances, nous nous apparentons aux travaux sur la machine abstraite YAM [MAN 08] qui implémente un langage de programmation pour manipuler des graphes orientés étiquetés et sur la machine abstraite AMaXoS [BRY 06] qui implémente un langage d'interrogation de données XML.

YAM implémente le langage de programmation de graphes GP basé sur des schémas de règles de transformation de graphes. YAM permet ainsi de réaliser des opérations quelconques de manipulation de graphes orientés étiquetés. KGRAM et YAM travaillent donc sur les mêmes objets, à savoir des nœuds et des arêtes de graphes mais YAM est une machine abstraite de très bas niveau tandis que nous avons recherché dans KGRAM un haut niveau d'abstraction : YAM manipule des tables de hachage tandis que KGRAM manipule des interfaces de nœuds et d'arêtes ; les instructions de YAM sont des opérations simples sur des piles tandis que le langage de KGRAM est une API d'interrogation de graphe de haut niveau — qui notamment généralise et étend SPARQL.

Amaxos implémente le langage Xcerpt pour l'interrogation des données semi-structurées du web. Dans ce langage, les requêtes sont des patrons et les réponses aux requêtes des instances de patrons. En ce sens, KGRAM et Amaxos partagent le même but d'unifier l'interrogation de données sur le web en subsumant à la fois les

1. http ://www-sop.inria.fr/edelweiss/software/corese/

langages de représentation des données et les langages d'interrogation de ces données. Cependant, dans la recherche des réponses à une requête, Amaxos ne considère que la structure des données, contrairement à KGRAM qui prend en compte la sémantique des nœuds et arcs des graphes de connaissances. Enfin, Amaxos exécute du code de bas niveau, résultat de la compilation de requêtes exprimées dans le langage de haut niveau Xcerpt. C'est donc comme YAM et contrairement à KGRAM une machine abstraite de bas niveau.

Le haut niveau d'abstraction que nous avons atteint dans la conception de KGRAM fait de celle-ci une plateforme générique. Sa généricité est tout d'abord relative aux différents langages de requêtes qu'elle interprète. GRAAL, son langage d'interrogation de graphes, est en effet conçu comme un ensemble extensible de primitives qui définissent une *famille* de langages. KGRAM permet donc d'interpréter différents langages de requêtes — moyennant l'écriture d'un compilateur pour un langage particulier. Cette tâche d'écriture d'un compilateur est facilitée par le niveau d'abstraction atteint dans KGRAM est tel que les structures de données manipulées sont des interfaces.

La généricité de KGRAM est également relative aux différents modèles de graphes de connaissances qu'elle permet d'interroger. Sa fonction d'évaluation d'une requête sur une base de graphes ne manipule que des interfaces — de nœuds et d'arcs de graphes — et fait appel à des fonctions d'interfaces — de gestionnaire abstrait de graphes, de comparateur abstrait de labels de nœuds, d'évaluateur abstrait de contraintes. KGRAM permet ainsi d'interroger des graphes de différents modèles — moyennant l'implémentation de ces interfaces. Cette généricité de KGRAM rend celle-ci interopérable en ce sens qu'elle lui permet d'exploiter des graphes issus de différents modèles en connectant différents gestionnaires de graphes et évaluateurs de contraintes implémentant les mêmes interfaces. Dans le plus simple des cas, KGRAM permet d'apparier des graphes orientés étiquetés — en réalisant une implémentation d'un comparateur de labels de nœuds rudimentaire. Comme nous le verrons dans la suite de cet article, nous avons réalisé des implémentations des interfaces de KGRAM permettant de prendre en compte la sémantique des graphes et ainsi d'apparier des graphes conceptuels avec contraintes et d'interroger des graphes RDF dans le langage SPARQL étendu.

La généricité et l'interopérabilité de KGRAM ouvrent la perspective de distribuer le traitement de requêtes sur différentes bases de graphes de connaissances qui peuvent être hétérogènes. Cette perspective est multiple. Tout d'abord, KGRAM peut être envisagé comme le moyen d'uniformiser l'interrogation de connaissances ou plus généralement de données à base de graphes dans différents modèles. Cela est un enjeu majeur sur le web de données où coexitent RDF/S, Topic Maps, XML, BD, etc. En outre, la combinaison de résultats partiels obtenus sur différentes bases permettrait de développer des applications composites ou *mashup*. Enfin, l'invocation par KGRAM de plusieurs gestionnaires de graphes dans des thread parallèles permettrait d'aborder le problème du passage à l'échelle.

Nous décrivons dans la partie 2 la syntaxe abstraite de GRAAL et sa sémantique opérationnelle en Sémantique Naturelle. La partie 3 est dédiée à la machine abstraite KGRAM que nous présentons comme l'interprète de GRAAL, c'est-à-dire l'implémentation des règles de sémantique naturelle de GRAAL. Nous montrons l'interopérabilité de KGRAM en la connectant indifféremment aux gestionnaires de graphes des moteurs sémantiques Corese et Jena[2] [MCB 02] et à l'évaluateur de contraintes de Corese. Nous concluons cet article et dressons les perspectives de notre travail dans la partie 4.

## 2. Le langage abstrait de requête GRAAL

Nous décrivons dans cette partie la syntaxe abstraite de GRAAL et sa sémantique opérationnelle et identifions deux sous-langages remarquables de GRAAL qui correspondent à celui des Graphes Conceptuels [SOW 84, CHE 09] et à SPARQL.

### 2.1. *Syntaxe abstraite de GRAAL*

La syntaxe abstraite de GRAAL est donnée par la grammaire suivante :

```
QUERY  ::= query(NODE *, EXP)
EXP    ::= QUERY | NODE | EDGE | FILTER | PATH
         | and(EXP, EXP) | union(EXP, EXP) | option(EXP)
         | not(EXP) | exist(EXP) | graph(NODE, EXP)
NODE   ::= node(label)
EDGE   ::= edge(label, NODE *)
PATH   ::= path(RegExp, NODE, NODE)
FILTER ::= filter(FilterExp)
```

Voici un exemple simple d'une requête GRAAL qui permet de rechercher les auteurs et les titres de documents (la requête ne dépend pas du modèle de graphes interrogés) :

```
query(
  {node('?x'), node('?title')},
  and(edge('hasCreated', {node('?x'), node('?doc')}),
      edge('hasTitle', {node('?doc'), node('?title')})))
```

Une requête est définie par une expression à évaluer et une liste de variables dont on cherche à exhiber les liaisons après l'évaluation de l'expression sur le graphe cible interrogé. Une expression QUERY permet d'exprimer une telle requête. Son paramètre EXP représente l'expression à évaluer et les paramètres NODE les variables dont on cherche les liaisons. Ces variables correspondent en syntaxe concrète à celles de la

2. http ://jena.sourceforge.net/

clause SELECT dans un langage "à la SQL" ou encore aux paramètres d'une lambda-expression dans le modèle des graphes conceptuels. Une expression QUERY permet également d'exprimer une requête imbriquée dans une autre. Dans ce cas, le résultat de son évaluation détermine des liaisons de variables pour le reste de l'expression requête contenante.

Les expressions NODE et EDGE permettent de rechercher un nœud ou une relation n-aire (hyperarc) dans un (hyper)graphe. Le paramètre `label` d'une expression NODE ou EDGE représente le label d'un nœud ou d'une relation dans un graphe ; c'est une constante (ou une variable pour NODE).

Le paramètre `FilterExp` d'une expression FILTER permet d'exprimer des contraintes sur les nœuds recherchés dans le graphe interrogé. C'est une expression booléenne d'un langage de contraintes (interprété par l'évaluateur de filtres donné à KGRAM) :

```
FilterExp ::= Variable | Constant | Term
Term      ::= Oper(FilterExp *)
Oper      ::= '<' | '<=' | '>=' | '=' | '!=' |
              '&' | '|' | '!' | '+' | '-' | '*' | '/' | FunctionName
```

Soulignons que les trois expressions NODE, EDGE et FILTER sont primitives et nous verrons dans la partie 3 qu'elles correspondent à des interfaces de la machine abstraite KGRAM qui interprète GRAAL.

Une expression PATH est une généralisation de l'expression EDGE et permet de rechercher un chemin de relations binaires entre deux nœuds dans un graphe. Le paramètre `RegExp` est une expression régulière décrivant un ensemble de chemins de relations. C'est le pendant, pour les graphes de connaissances, des chemins XPath pour les arbres XML. Un chemin élémentaire est une relation. Un chemin peut également être la répétition d'une même relation ou chemin de relations (représentée avec l'opérateur *) un nombre quelconque de fois (possiblement nul) ou une séquence de relations ou chemins de relations (représentée avec l'opérateur /) avec des alternatives possibles (représentées avec l'opérateur |) :

```
RegExp ::= label | RegExp '*' | RegExp'/'RegExp | RegExp'|'RegExp
```

Voici un exemple de requête permettant de retrouver tous les éléments d'une liste :

```
query({node('?y')}, path(rdf:rest*/rdf:first, node('?x'), node('?y')))
```

Une expression `and` (resp. union) permet d'exprimer une conjonction (resp. disjonction) entre deux expressions.

Une expression option rend facultative l'existence de solutions à une expression dans la recherche des solutions d'une requête.

Une expression `not` exprime la négation par l'échec.

Une expression `exist` permet de ne rechercher qu'*une* solution (la première trouvée).

Une expression `graph` permet de spécifier le graphe de connaissances sur lequel évaluer une expression (en l'absence d'une telle expression, c'est un graphe par défaut qui est considéré).

## 2.2. *Sémantique naturelle de GRAAL*

La Sémantique Naturelle a été initialement conçue par [KAH 87] pour fournir une sémantique opérationnelle aux langages de programmation, les règles de sémantique naturelle constituant les spécifications des interprètes de ces langages. De manière analogue, en définissant une sémantique opérationnelle de GRAAL, nous spécifions la machine abstraite KGRAM qui peut être vue comme un interprète de GRAAL, les expressions de celle-ci permettant d'interroger des bases de graphes de connaissances.

### 2.2.1. *Principes de Sémantique Naturelle*

En Sémantique Naturelle, la sémantique opérationnelle d'un langage est donnée sous la forme d'un ensemble de règles d'inférence, où les expressions du langage sont évaluées dans un environnement et l'application d'une règle d'inférence produit une liste d'environnements. Ainsi, les règles d'inférence de la sémantique de GRAAL décrivent l'évolution de l'environnement — initialement vide — lors de l'évaluation des expressions du langage qui composent une requête. Plus précisément, une expression dans une requête est évaluée dans un environnement qui contient une valuation des variables apparaissant dans la requête et liées à des nœuds du graphe interrogé, ces liaisons résultant de l'évaluation précédente d'autres expressions de la requête. L'évaluation d'une expression peut produire plusieurs environnements (dans le cas de plusieurs solutions) et les autres expressions de la requête évaluées ensuite le sont dans chacun d'eux. Lorsque toutes les expressions d'une requête ont été évaluées, chaque environnement qui en résulte correspond à une solution trouvée.

Formellement, les règles 1 et 2 suivantes décrivent l'évolution de l'environnement lors de l'évaluation d'une expression GRAAL. Une liste d'environnements est décrite par son premier élément ENV et la liste LENV de ses autres éléments. L'évaluation d'une expression EXP dans chacun des environnements de la liste ENV.LENV produit une liste d'environnements LENV'.LENV" où LENV' est le résultat de l'évaluation de EXP dans ENV et où LENV" est le résultat de l'évaluation de EXP dans chacun des environnements de LENV (en appliquant récursivement les règles 1 et 2).

$$\frac{ENV \vdash EXP \rightarrow nil \ \wedge \ LENV \vdash EXP \rightarrow LENV'}{ENV \,.\, LENV \vdash EXP \rightarrow LENV'} \qquad [1]$$

$$\frac{ENV \vdash EXP \rightarrow LENV'' \ \wedge \ LENV \vdash EXP \rightarrow LENV'}{ENV \,.\, LENV \vdash EXP \rightarrow LENV'' \,.\, LENV'} \qquad [2]$$

2.2.2. *Node et edge*

Les règles 3 et 4 suivantes montrent le calcul des listes d'environnements lors de l'évaluation d'une expression demandant la recherche d'un nœud NODE ou d'une relation EDGE dans un graphe. L'évaluation d'une telle expression dans un environnement *ENV* requiert de rechercher la liste d'environnements *LENV* contenant les appariements possibles de NODE ou de EDGE dans le graphe interrogé et de fusionner *ENV* et *LENV*. Ces deux opérations sont synthétisées par les bases de règles *match* et *merge* que nous ne décrivons pas ici. Ces bases définissent la sémantique respectivement du comparateur d'étiquettes de nœuds et d'arcs et du gestionnaire d'environnements de KGRAM (cf. partie 3).

$$\frac{match(ENV \vdash NODE \rightarrow LENV) \ \wedge \ merge(ENV, LENV \rightarrow LENV')}{ENV \vdash NODE \rightarrow LENV'} \quad [3]$$

$$\frac{match(ENV \vdash EDGE \rightarrow LENV) \ \wedge \ merge(ENV, LENV \rightarrow LENV')}{ENV \vdash EDGE \rightarrow LENV'} \quad [4]$$

2.2.3. *Filter*

Les règles 5 et 6 suivantes montrent que la base de règles *eval* relative à l'évaluation de l'expression booléenne paramètre d'une expression FILTER utilise les liaisons des variables de la requête dans l'environnement courant. Si cette expression est évaluée à faux (règle 5), une liste d'environnements vide est produite (nil) : il n'y a pas de solution ; sinon (règle 6), la liste produite contient comme unique élément l'environnement courant (elle est créée à partir de celui-ci avec l'opérateur *list*).

$$\frac{eval(ENV \vdash F : false)}{ENV \vdash filter(F) \rightarrow nil} \quad [5] \qquad \frac{eval(ENV \vdash F : true)}{ENV \vdash filter(F) \rightarrow list\ ENV} \quad [6]$$

2.2.4. *And et Union*

La règle 7 suivante montre que l'évaluation de l'une des expressions paramètres d'une expression *and* dans l'environnement courant produit une liste d'environnements dans laquelle est évaluée la seconde expression paramètre et la liste d'environnements produite par cette évaluation est le résultat de l'évaluation de l'expression *and*. La règle 8 montre que dans le cas où la première expression évaluée produit un environnement vide (il n'y a pas de réponse à cette expression requête), il est inutile d'évaluer la seconde (il n'y a pas de réponse à l'expression *and*).

$$\frac{ENV \vdash A \rightarrow LENV \ \wedge \ LENV \vdash B \rightarrow LENV'}{ENV \vdash and(A, B) \rightarrow LENV'} \quad [7]$$

$$\frac{ENV \vdash A \rightarrow nil}{ENV \vdash and(A, B) \rightarrow nil} \quad [8]$$

La règle 9 suivante montre que l'environnement produit par l'évaluation d'un expression *union* dans un environnement *ENV* est la juxtaposition de ceux produits par l'évaluation de chacune des deux sous-expressions de l'expression *union* dans *ENV*.

$$\frac{ENV \vdash A \rightarrow LENV \ \wedge \ ENV \vdash B \rightarrow LENV'}{ENV \vdash union(A,B) \rightarrow LENV \ . \ LENV'} \quad [9]$$

### 2.2.5. *Option*

Les règles 10 et 11 suivantes montrent que si l'évaluation d'une expression *option(A)* réussit (règle 10), l'évaluation du reste de la requête tient compte des liaisons de variables issues de l'évaluation de *option(A)*. Sinon (règle 11), l'environnement considéré dans le reste de la requête est tel qu'il était avant l'évaluation de *option(A)*.

$$\frac{ENV \vdash A \rightarrow LENV}{ENV \vdash option(A) \rightarrow LENV} \quad [10] \qquad \frac{ENV \vdash A \rightarrow nil}{ENV \vdash option(A) \rightarrow list\ ENV} \quad [11]$$

### 2.2.6. *Not*

Les règles 12 et 13 suivantes stipulent que l'évaluation d'une expression *not(A)* produit un environnement solution lorsque l'évaluation de l'expression *A* produit un environnement vide et inversement.

$$\frac{ENV \vdash A \rightarrow nil}{ENV \vdash not(A) \rightarrow list\ ENV} \quad [12] \qquad \frac{ENV \vdash A \rightarrow LENV}{ENV \vdash not(A) \rightarrow nil} \quad [13]$$

### 2.2.7. *Exist*

Les règles 14 et 15 stipulent que l'environnement résultat de l'évaluation d'une expression *exist(A)* est vide dans le cas où le résultat de l'évaluation de *A* est vide et que dans le cas contraire l'environnement courant reste inchangé.

$$\frac{ENV \vdash A \rightarrow nil}{ENV \vdash exist(A) \rightarrow nil} \quad [14] \qquad \frac{ENV \vdash A \rightarrow LENV}{ENV \vdash exist(A) \rightarrow list\ ENV} \quad [15]$$

### 2.2.8. *Query*

La règle 16 suivante stipule que pour évaluer une expression *query(LNODE, EXP)*, les liaisons des nœuds en paramètre de l'expression sont extraites de l'environnement courant *ENV* : la base de règles *select* synthétise cette opération qui n'est pas détaillée ici. Puis l'expression *EXP* est évaluée dans l'environnement *ENV'* ainsi produit. Les liaisons des nœuds en paramètre de l'expression *EXP* sont alors extraites de l'environnement *LENV'* résultat de cette évaluation (selon la même base de règles *select*) et sont fusionnées avec l'environnement initial *ENV* (selon la base de règles *merge* déjà rencontrée dans les règles 3 et 4). Ainsi, une expression *query* "emboitée" dans une autre ne partage avec cette dernière que les nœuds qui figurent dans sa liste de nœuds en paramètre.

$$\frac{select(ENV \vdash LNODE \rightarrow ENV') \wedge ENV' \vdash EXP \rightarrow LENV' \wedge \\ select(LENV' \vdash LNODE \rightarrow LENV'') \wedge merge(ENV, LENV'' \rightarrow LENV)}{ENV \vdash query(LNODE, EXP) \rightarrow LENV} \quad [16]$$

2.2.9. *Graph*

L'introduction de l'expression *graph(NODE, EXP)* dans KGRAM requiert, sur le modèle de la règle 17 suivante relative à cette expression, d'ajouter un argument *G* à toutes les autres règles de sémantique du langage (nous n'avons délibérément pas ajouté cet argument dans la présentation des règles qui est faite dans l'article par souci de simplification). Cet argument est un nœud qui représente le nom du graphe cible courant à interroger c'est-à-dire sur lequel évaluer une expression. Il est pris en compte par la base de règles *match* (règles 3 et 4) qui sélectionne ainsi le graphe courant sur lequel rechercher des appariements. Ce peut être une constante ou une variable pour laquelle le paquet de règles *match* détermine alors les liaisons.

La règle 17 stipule que le résultat de l'évaluation d'une expression *graph(G', EXP)* est celui de l'évaluation de l'expression *EXP* sur le graphe *G'*.

$$\frac{ENV, G' \vdash EXP \rightarrow LENV}{ENV, G \vdash graph(G', EXP) \rightarrow LENV} \quad [17]$$

2.2.10. *Path*

La règle 18 est relative à l'évaluation d'une expression *path* avec en paramètre le motif élémentaire de chemins de relations : une relation *P* entre deux nœuds *N1* et *N2*. Il s'agit dans ce cas d'évaluer une expression *edge* (en appliquant la règle 4).

$$\frac{ENV \vdash edge(P, N1, N2) \rightarrow LENV}{ENV \vdash path(P,\ N1,\ N2) \rightarrow LENV} \quad [18]$$

La règle 19 suivante stipule que le résultat de l'évaluation d'une expression de la forme *path(EXP1/EXP2, N1, N2)* s'obtient en évaluant dans l'environnement courant une expression *path* avec en paramètres l'expression régulière *EXP1* et le nœud *N1* comme source du chemin puis en évaluant dans l'environnement résultat, une expression *path* avec en paramètres l'expression régulière *EXP2* et le nœud *N2* comme cible.

$$\frac{ENV \vdash path(EXP1,\ N1,\ Ni) \rightarrow LENV \wedge \\ LENV \vdash path(EXP2,\ Ni,\ N2) \rightarrow LENV'}{ENV \vdash path(EXP1\ /\ EXP2,\ N1,\ N2) \rightarrow LENV'} \quad [19]$$

La règle 20 stipule que l'environnement produit par l'évaluation d'une expression *path(EXP1|EXP2, N1, N2)* est la juxtaposition des environnements résultats de l'évaluation dans l'environnement courant de deux expressions *path* ayant en paramètres

l'une l'expression régulière *EXP1* et l'autre celle *EXP2* et les mêmes nœuds source et cible.

$$\frac{\begin{array}{c} ENV \vdash path(EXP1,\ N1,\ N2) \rightarrow LENV \wedge \\ ENV \vdash path(EXP2,\ N1,\ N2) \rightarrow LENV' \end{array}}{ENV \vdash path(EXP1 \mid EXP2,\ N1,\ N2) \rightarrow LENV.LENV'} \quad [20]$$

Les règles 21 et 22 décrivent l'évaluation d'une expression *path(EXP*, N1, N2)*. Le cas général est décrit par la règle 21 : il s'agit d'évaluer une expression *path* avec en paramètre une occurence du motif *EXP* entre *N1* et un nœud intermédiaire *Ni* et dans l'environnement produit d'évaluer une expression *path* avec en paramètres *Ni* comme source et *N2* comme cible. Cette règle s'applique récursivement jusqu'à la condition d'arrêt décrite par la règle 22. Celle-ci décrit le cas où il n'y a pas de chemin de relations entre les nœuds *N1* et *N2*. Elle stipule d'ajouter à l'environnement courant les liaisons du nœud *N1* (en appliquant la règle 3) puis d'ajouter les liaisons du nœud *N2* qui coïncident avec celles de *N1*. Cette dernière opération est synthétisée par la base de règles *bind* que nous ne décrivons pas ici.

$$\frac{\begin{array}{c} ENV \vdash path(EXP,\ N1,\ Ni) \rightarrow LENV \wedge \\ LENV \vdash path(EXP*,\ Ni,\ N2) \rightarrow LENV' \end{array}}{ENV \vdash path(EXP*,\ N1,\ N2) \rightarrow LENV'} \quad [21]$$

$$\frac{ENV \vdash N1 \rightarrow LENV \wedge bind(LENV \vdash N2, N1 \rightarrow LENV')}{ENV \vdash path(EXP*,\ N1,\ N2) \rightarrow LENV'} \quad [22]$$

### 2.3. *Des sous-langages de GRAAL remarquables*

Selon le sous-ensemble des expressions de GRAAL que l'on considère, on définit un sous-langage de requêtes particulier ou un autre. Notamment, les expressions NODE et EDGE définissent un sous-langage de GRAAL correspondant à celui des Graphes Conceptuels simples [SOW 84, CHE 09]. L'opérationalisation des règles de sémantique naturelle associées à ces expressions correspond à la recherche d'homomorphismes de graphes étiquetés dont les relations peuvent être n-aires. Nous verrons dans la partie 3 que ce calcul d'homomorphismes de graphes est la clé de voûte de l'algorithme de KGRAM. En ajoutant au langage des expressions FILTER, on considère le modèle des Graphes Conceptuels *avec contraintes* [BAG 02].

Les expressions NODE, EDGE, FILTER, *and()*, *union()*, *option()* et *graph()* définissent un sous-langage de GRAAL qui correspond au fragment cœur du langage SPARQL (étendu aux relations n-aires) défini par le schéma de requête SELECT-WHERE et les opérateurs UNION, AND, OPTIONAL, FILTER et GRAPH. En outre, l'expression *exist()* de GRAAL permet d'exprimer le schéma de requête ASK du langage SPARQL et la notion de requête imbriquée que modélise l'expression *query()* est un des sujets actuels du groupe de travail SPARQL 1.1.

## 3. La machine abstraite KGRAM

La machine abstraite KGRAM est conçue comme un interprète de GRAAL : nous présentons dans cette partie sa fonction d'évaluation d'une expression GRAAL qui opérationalise les règles de sémantique naturelle du langage. Nous commençons par décrire l'interface de programmation que manipule l'algorithme.

### 3.1. *Interfaces KGRAM*

KGRAM accède au graphe interrogé au travers d'une interface abstraite (API) qui en masque la structure et l'implémentation. Autrement dit, KGRAM opère sur une abstraction de graphe, au travers de structures et de fonctions abstraites et ignore la structure interne des sommets et hyperarcs qu'il manipule dans sa fonction d'évaluation d'une expression GRAAL sur un graphe cible. Plus précisément, le graphe interrogé est modélisé sous forme de piles de sommets et d'hyperarcs d'arité n quelconque qui implémentent les interfaces *Node* et *Edge*. D'autre part, ces mêmes interfaces opérationalisent les expressions NODE et EDGE du langage de requête GRAAL. Ainsi KGRAM permet d'interroger tout type de graphe de connaissances, par exemple aussi bien des graphes conceptuels (dont les relations sont n-aires) que des graphes RDF (dont les relations sont binaires).

Non seulement les structures de données manipulées par KGRAM sont abstraites, mais aussi ses opérateurs :

– Le gestionnaire de graphes de KGRAM qui permet d'accéder au graphe interrogé est un objet implémentant l'interface *Producer* qui énumère les *Node* et *Edge* du graphe qui s'apparient aux *Node* et *Edge* de la requête. Il fait appel à l'API du comparateur de sommets et d'arcs et à celle du gestionnaire d'environnements de KGRAM et ignore donc la façon dont les sommets et les arcs sont appariés.

– Le comparateur de sommets et d'arcs est un objet implémentant l'interface *Matcher*. Il a la charge de la comparaison des étiquettes (labels et types) de sommets et d'arc. Il implémente le paquet de règles *match* utilisé dans les règles 3 et 4 de la sémantique naturelle de GRAAL. Selon l'implémentation de *Matcher*, la comparaison des étiquettes consistera en un simple test d'égalité des labels ou bien prendra en compte les relations de subsomption entre types, ou encore autorisera des appariements approchés basés sur des mesures de similarité, etc.

– Les contraintes (ou filtres) sont des objets abstraits qui implémentent l'interface *Filter*. Cette interface correspond à l'expression FILTER de GRAAL. Les filtres sont évalués par un objet qui implémente l'interface *Evaluator*. KGRAM ignore la structure interne des filtres qu'il manipule, il se contente d'appeler la fonction `eval` d'*Evaluator* sur des objets *Filter*, en passant en argument un *Environment*. Cette fonction eval implémente le paquet de règles *eval* utilisé dans les règles 5 et 6 de la sémantique naturelle de GRAAL.

En somme, KGRAM implémente les expressions de GRAAL et ses règles de sémantique naturelle dans une API, ce qui met en lumière le haut niveau d'abstraction auquel nous nous sommes tenus dans la conception de KGRAM. L'algorithme d'évaluation d'une requête de KGRAM qui ne manipule que ces interfaces est ainsi totalement abstrait, indépendant de toute implémentation et de toute structure de données.

### 3.2. *Algorithme de KGRAM*

L'évaluation d'une expression GRAAL correspond à la recherche des réponses qui existent à une requête dans la base de graphes interrogée. Elle repose sur l'opérationalisation des règles 3 et 4 de sémantique naturelle associées aux expressions NODE et EDGE de GRAAL. Les environnements produits par ces règles représentent les homomorphismes trouvés entre l'expression GRAAL évaluée et le(s) graphe(s) interrogé(s). L'algorithme de KGRAM pour l'évaluation d'une expression GRAAL est décrit ci-après. Le paramètre `queryStack` de la fonction `eval` désigne la pile des expressions qui composent une expression GRAAL à évaluer et `i` représente un curseur dans cette pile. La fonction est initialement appelée avec une valeur de `i` nulle. Une instance de KGRAM est munie lors de sa création d'un `producer` qui implémente l'interface *Producer*, d'un `matcher` qui implémente l'interface *Matcher*, d'un `evaluator` qui implémente l'interface *Evaluator*, d'un gestionnaire d'environnement `env` stockant dans des structures de pile l'environnement courant d'évaluation — c'est-à-dire un homomorphisme partiel qu'il s'agit de compléter — et enfin d'une liste dans laquelle sont stockés les homomorphismes complets.

```
eval(queryStack, i){
  if (queryStack.size() = i){
    store(env); return;}
  exp = queryStack(i);
  switch(exp){
    case EDGE:
      for (Edge r : producer.candidate(exp, env)){
           env.push(exp, r))
           eval(queryStack, i+1);
           env.pop(exp, r);}
      break;
    case NODE:
      for (Node n : producer.candidate(exp, env)){
           env.push(exp, n))
           eval(queryStack, i+1);
           env.pop(exp, n);}
      break;
    case FILTER:
      if (evaluator.eval(exp, env)) eval(queryStack, i+1);
  }}
```

Dans l'instruction de contrôle `switch`, les blocs d'instructions étiquetés par les types NODE et EDGE comme valeurs de la variable `exp` implémentent les règles 3 et 4 et donc augmentent l'environnement courant par des liaisons des nœuds de la requête avec des nœuds du graphe interrogé. Pour ce faire, la fonction `candidate` du gestionnaire de graphes `producer` est appelée, qui prend en paramètres une expression NODE ou EDGE dans la pile `queryStack` des expressions de la requête et l'environnement courant `env`. Elle utilise l'environnement pour trouver les éventuels nœuds dans l'expression `exp` déjà liés, de sorte qu'elle retourne les seules relations compatibles avec les liaisons dans l'environnement. Ces relations sont à tour de rôle ajoutées dans l'environnement courant. La recherche d'homomorphisme aboutit et l'homomorphisme partiel en construction est complet lorsque le sommet de la pile `queryStack` est atteint : *env* est alors ajouté dans la liste de solutions en invoquant la fonction `store()`.

Lorsque l'homomorphisme partiel que constituait l'environnement courant ne peut être complété, le mécanisme de retour arrière ou *backtrack* dans la pile des appels récursifs de la fonction `eval` permet de revenir à un état antérieur de la pile des expressions de la requête et ce faisant à un état antérieur de l'environnement courant associé dans lequel de nouvelles liaisons peuvent être considérées pour certain(s) sommets(s) ou arc(s).

Ainsi, la double récursion que suggèrent les règles 3 et 4 de la sémantique naturelle de GRAAL — sur l'expression à évaluer et sur la liste d'environnements — se traduit dans l'algorithme de la fonction `eval` de KGRAM par d'une part des appels récursifs de la fonction `eval` et d'autre part le stockage itératif des homomorphismes complets construits dans `env`.

Le bloc d'instructions étiqueté par le type FILTER comme valeur de la variable de contrôle `exp` opérationalise les règles 5 et 6 de sémantique naturelle des expressions FILTER de GRAAL et KGRAM implémente ainsi la recherche d'homomorphismes de graphe sous contraintes. L'algorithme de KGRAM utilise un évaluateur de filtres abstrait `evaluator` et reste indépendant de la nature des filtres traités — qui dépendent du langage de filtres implémenté par l'évaluateur appelé. La fonction d'évaluation de l'évaluateur de filtres prend en argument le filtre à évaluer et l'environnement courant d'évaluation. Dans le cas où le filtre est évalué à vrai, la recherche d'homomorphisme continue avec l'environnement de l'évaluation du filtre. Dans le cas contraire, l'homomorphisme partiel que constituait l'environnement courant n'aboutit pas à une solution et un mécanisme de retour arrière dans la pile des expressions de la requête est mis en œuvre qui permettra de revenir à l'évaluation du filtre avec d'autres environnements courant.

L'ensemble des règles de sémantique naturelle de GRAAL sont opérationalisées dans KGRAM en intégrant à la "colonne vertébrale" de l'algorithme présentée ci-dessus de nouveaux blocs dans l'instruction de contrôle `switch`, pour traiter chaque type possible pour la variable de contrôle `exp`. Nous ne les détaillons pas dans cet article. Notons simplement que ce sont certains de ces blocs d'instructions qui modifient la pile `queryStack` d'expressions GRAAL à évaluer.

### 3.3. *Interopérabilité de KGRAM*

Nous avons testé la portabilité de KGRAM en réalisant une implémentation des interfaces *Node*, *Edge*, *Producer*, *Matcher* et *Evaluator* de KGRAM avec des composants de Corese d'une part et de Jena d'autre part. Pour valider KGRAM et son portage sur une implémentation ou une autre, nous utilisons une base d'environ 500 requêtes et une base de graphes avec environ 25000 arcs.

Les interfaces de KGRAM ont été conçues pour minimiser le "code glue" à réaliser de sorte que les maquettes de portage de KGRAM sur un moteur ou sur l'autre ont demandé relativement peu de développement.

La connexion de Corese à KGRAM a été presque immédiate, ce qui s'explique par le fait que nous avons conçu KGRAM en abstrayant les principes de Corese. Avec Corese, KGRAM interprète l'ensemble des expressions de GRAAL et peut interroger des graphes R§DF ou des graphes conceptuels. Avec Jena, le portage a nécessité l'écriture de 416 lignes de code, soit quatre classes : `NodeImpl`, `EdgeImpl`, `ProducerImpl` et `EvaluatorImpl` (KGRAM utilise dans ce cas un matcher par défaut). Avec Jena, KGRAM peut interroger des graphes RDF ; il interprète actuellement des expressions *edge*, *filter*, *union*, *and* et *optional*. Ce travail de portage se poursuit afin de prendre en compte l'ensemble des primitives de KGRAM correspondant à SPARQL.

Ces deux différentes implémentations témoignent de la généricité de la conception de KGRAM et laissent supposer une connexion facile d'autres implémentations de gestionnaires de graphes de connaissances avec KGRAM.

## 4. Conclusion et perspectives

Nous avons présenté dans cet article la machine abstraite KGRAM pour l'interrogation de graphes de connaissances et son langage de requête à base de graphes GRAAL. Nous avons établi des règles de sémantique naturelle pour chacune des expressions de GRAAL, ces règles d'inférence constituant les spécifications de KGRAM qui les opérationalise et qui peut être vu comme un interprète du langage GRAAL. Nous avons mis en lumière le niveau d'abstraction de KGRAM, la simplicité de son algorithme reposant sur la manipulation d'*interfaces* aussi bien pour les opérateurs que pour les structures de données.

Dans la continuité des résultats présentés dans cet article, nous travaillons actuellement à l'intégration d'optimisations telles que celles proposées par [COR 07], comme le tri heuristique des arcs de la requête ou encore la possibilité de demander plusieurs relations adjacentes au gestionnaire de graphes plutôt qu'il les énumère.

Nous prévoyons de travailler à l'implémentation des interfaces KGRAM pour interroger des connaissances dans le modèle des Topic Maps et des données dans des modèles plus simples tels que XML ou GraphML. A terme, nous envisageons de poursuivre notre travail de définition du langage GRAAL et de spécification d'interfaces

KGRAM en étudiant les possibilités de généralisation et d'extension pour interroger des connaissances dans des modèles à base de frames et des logiques de description.

Enfin, nos perspectives de travail sont d'aborder le problème de la distribution du traitement du web de données en interconnectant différents gestionnaires de graphes implémentant chacun l'API de KGRAM et responsables chacun d'une base de connaissances dans des modèles possiblement différents. Nous envisageons KGRAM d'une part comme un élément de réponse au problème du passage à l'échelle dans le traitement du web de données et d'autre part comme la clé de voute d'applications composites (ou *mashup*) combinant les résultats de différents gestionnaires de graphes.

## 5. Bibliographie

[BAG 02] BAGET J., MUGNIER M., « Extensions of Simple Conceptual Graphs : the Complexity of Rules and Constraints », *J. Artif. Intell. Res. (JAIR)*, vol. 16, 2002, p. 425-465.

[BAG 08] BAGET J., CORBY O., DIENG-KUNTZ R., FARON-ZUCKER C., F.GANDON, GIBOIN A., GUTIERREZ A., LECLÈRE M., MUGNIER M., THOMOPOULOS R., « GRIWES : Generic Model and Preliminary Specifications for a Graph-Based Knowledge Representation Toolkit », *Proc. of the 16th International Conference on Conceptual Structures, ICCS 2008*, vol. 5113 de *Lecture Notes in Computer Science*, Springer, 2008, p. 297-310.

[BRY 06] BRY F., FURCHE T., LINSE B., « AMaXoS Abstract Machine for Xcerpt : Architecture and Principles », *Proc. of 4th Workshop on Principles and Practice of Semantics Web Reasoning*, vol. 4187 de *Lecture Notes in Computer Science*, 2006, p. 105–119.

[CHE 09] CHEIN M., MUGNIER M., *Graph-based Knowledge Representation : Computational Foundations of Conceptual Graphs*, Springer London Ltd, 2009.

[COR 04] CORBY O., DIENG-KUNTZ R., FARON-ZUCKER C., « Querying the Semantic Web with Corese Search Engine », *Proc. of the 16th Eureopean Conference on Artificial Intelligence, ECAI 2004*, IOS Press, 2004, p. 705-709.

[COR 07] CORBY O., FARON-ZUCKER C., « Implementation of SPARQL Query Language Based on Graph Homomorphism », *Proc. of the 15th International Conference on Conceptual Structures, ICCS 2007*, vol. 4604 de *Lecture Notes in Computer Science*, Springer, 2007, p. 472-475.

[KAH 87] KAHN G., « Natural Semantics », *Proc. of 4th Annual Symposium on Theoretical Aspects of Computer Science, STACS 87*, vol. 247 de *Lecture Notes in Computer Science*, Springer, 1987, p. 22-39.

[MAN 08] MANNING G., PLUMP D., « The York Abstract Machine », *Electron. Notes Theor. Comput. Sci.*, vol. 211, 2008, p. 231–240, Elsevier Science Publishers B. V.

[MCB 02] MCBRIDE B., « Jena : A Semantic Web Toolkit », *IEEE Internet Computing*, vol. 6, n° 6, 2002, p. 55-59.

[SOW 84] SOWA J., *Conceptual Structures : Information Processing in Mind and Machine*, Addison-Wesley, 1984.